Tuesday, 29 November 2022

Laravel

* We can use php artisan serve to start the laravel app at port 8000 of localhost. To specify an alternate port we can use ‘php artisan serve —port:portnumber’ to start the app at some other port number you want.
* Another way of starting the app is with vagrant and virtual box which is very complicated. Homestead lets you have an extension for your project in the browser.

**Important:- The routes are configured at Routes folder in web.php in the app.**

**Laravel Structure**

* App folder is the main folder
* Bootstrap is the folder where bootstrap is loaded into the application
* Configuration folder is the folder you place all your configuration such as database
* Inside configuration folder mail.php folder is used to configure the mail server
* In configuration/migrations we can create tables.
* In configuration/factories we can use it for testing
* Public folder has all the public files such as html css etc
* Resources folder is used to store compiled files. It is done using web pack.
* In Resources/views we store the views of the application
* After compilation of resources it will go to resources folder from views folder
* Api’s and endpoints are created in api.php(apis which are not public)
* Vendor is the folder that is created by composer
* .env file is used to store sensitive information such as db details such as username, password
* Put .env in .gitignore
* Artisan is an application which runs everything such as creating classes etc.
* Composer.json
* Packages-json for node js.
* server.php wraps the built in php server. It emulates apache’s “mod rewrite” functionality.
* Webpack is a wrapper which wraps the functionalities and compile .

**Routes:**

* It is what is on the url of the websites.
* Route group is the collection of routes.
* Middleware is a secure feature inside the routes of laravel.
* Route class
* [::indicate that it is astatic method]
* Eg;-Route::get(‘/‘,function(){

return view(‘welcome’);

}); / - takes the root as path

The function is called closure function.

Instead of function we can use return a text.

* You can pass parameters to the route through url

Eg;- Route::get(‘/post/{id}/{name}’,function($id,$name)

return “Hi this “.$post.” “.$name;

);

In the url we can pass these parameters.

Naming Routes:-

If the path of the route is too long it is difficult for the user to type the entire url. So we can use route naming to solve the problem.

For this use the second parameter of the get method.

Eg:-

Route::get(‘/post/admin/example’,array(‘as’=>’admin.home’,function(){

$url = route(‘admin.home’);

return “this url is “.$url;

}));

route() inside the body of get method is a helper function.

If you give php artisan route:list, you can see a list of routes in which you can see the name of the route of as admin.home.

Instead of using the entire url in our code we can use it by just calling the name we defined.

**Controllers**

* It is under app/http/controllers.
* They act as intermediary between the views and databases or other representational models.
* Namespace keyword is a special keyword used in controllers. Namespace in a class is similar to scope of a function. It is used to avoid collation when creating identifiers of same name. Namespaces once created can include various functionalities which can be used in controllers and various classes.
* Use is to import a specific class or namespace to the current file.
* To create a controller you can create a php file under controller.
* Or you can use terminal command

Php artisan make:controller controller\_name

Eg:- php artisan make:controller Post\_Controller

* another way is

Php artisan make:controller —resource PostController

This creates a crud operation controller. Which has an index(),store(),show(),update(),destroy() methods.

**Routing Controllers**

* Note : From Laravel 8 we have to give the full path of the controller in the route

ie:- Route::get(‘/checking’,’\App\Http\Controllers\HomeController@index’);

Or

Use App\Http\Controllers\HomeController;

Route::get(‘/‘,[EdwinsController::class,’index’]);

* In routes if we want to use a controller instead of a closure function we can just give the entire path to the controller as the second parameter for the get method.
* We also need to specify which method we should use in the controller so we have specify it with @method\_name which is in the controller.
* Eg:- Route::get(‘/post’,’App\Http\Controllers\PostController@index’)
* To pass data to the controller from route we can use

Eg:- Route::get(‘posts/{id}’,’\App\Http\Controllers\PostController@index’)

In controller we can accept the parameter.

**Resources and Controller**

* Resource is a special static function like get that will give special types of routes automatically.
* Eg: - Route::resource(‘posts’,’PostController’); By just specifying this we get access to index, store, update, destroy methods.
* To show all the methods of the route use php artisan route:list which will list all the routes and methods.
* If you want to access the show method, in the url give posts/1 or posts/2 or posts/any number which is the id that is accepted by the show method.

**Views**

* Views are located in the resources folder
* To create a view we can create a php file under resources/views.
* The file name should be file\_name.blade.php. Because Laravel uses blade templating engine.
* Inside the view add any content you want such as html tags etc. Then in the controller create a function to return this view.

Eg:- function view\_contact(){

return view(‘contact’);

}

This will return the contact view page. Just make sure that the controller method is added to the routes. Ie, in routes

Route::get(‘/contact’,’\App\Http\Controllers\PostController@view\_contact’);

**Passing Data into views**

* One way to pass data to the view is through route. From the route’s url we can accept a data as a variable. The same variable is passed to the controller method as an argument. From controller when the view is called we can chain it with ->with(‘name\_of\_variable’,variable);

Eg:-

Route::get(‘/posts/{id}’,’\App\Http\Controllers\PostController@show\_post’);

in the show\_post() method

function show\_post($id){

return view(‘showpost’)->with(‘id’,$id);

}

we can use the id variable in the view by {{$id}}

* Another way is to use compact() inside the calling of view function in the controller

Ie, return view(‘showpost’,compact(‘id’));

This let’s you pass any number of arguments to the view. The only thing to note here is that the parameter you are passing to the compact method should be in single quotes and the name should be the same as the variable name which is present in the argument of the controller function.

* You can pass multiple parameters to the view with compact() in a ‘,’ separated way.
* You can also pass arrays with compact()

**BladeTemplating Engine**

-It makes php code easier to use in html pages.

- We can create a master template inside views eg:- inside views layouts/app.blade.php. Create an html body inside which you can put

**@yield(’content’)**

And @yield(‘footer’)

Inside the particular html view where you want to extend the master template

Use @extends(’layouts.app’)

Which extends the app template from layouts folder.

To put content in the @section(’content’)

Note that name of the section should be same as the name defined in @yield of the master template.

To close the section you can use @endsection or @stop

Between the section tag and end section tag put the html tags which you want to use.

* It is not mandatory to use all the yields as sections defined in master template. We can use what we need whenever we need.
* Note () for yield, section and extends.
* @if () ….@endif give same functionality as if statement in php inside blade.
* @foreach @endforeach is the same as foreach loop in php for blade.
* @include is used to add files in blade.
* Example:-

in our controller we have

$people = [‘Alex’,’Jose’,’John’,’jack’];

we can send this data to view show\_contact by

return view(‘contact’,compact(‘people’));

in the contact.blade.php

we use

@if(count($people))

<ul>

@foreach($people as $person)

<li>{{$person}}</li>

@endforeach

</ul>

@endif

**Migrations**

* Composer creates .env and .env.example files automatically for your projects.
* .env files is the file which is used to store all the environment variables such as configuration related data such as database password, api keys etc.
* We don’t want this sensitive information to be available to others. The .env is not sharable so in-order to share the necessary configurations .env.example file is used. While sharing the project .env.example is shared and in-order for the project to work in the other person’s system they have to rename .env.example to .env.
* Sqlite is file based database, which means we don’t need special programs to run a database. It is used for light weight applications. All the data is stored in files.
* Migration is a feature of Laravel which let’s automate things and certain database configurations for us.
* By default in the .env file the configuration for mysql is set to localhost server at port 3306 with user name as root. We can change it if we want or we change the database name to which we are connecting.
* After editing the .env file if we type “php artisan migrate”, it will automatically create tables which is defined in the files inside database/migrations/
* The Schema class is used for database operations. The Schema class has static methods such as create.
* The create function helps to create table. As an argument to the function we specify the name of the table. In the function body we specify the columns.
* To create a migration we can use the php artisan command line tool

php artisan make:migration migration\_name - -create=“table\_name”

This will generate some boilerplate code for you.

* In the generated code we can make changes in-order to add columns. We can also specify constraints for these columns. For further information about this visit Laravel documentation.
* After making the necessary changes to the migration file in database/migrations folder we can use php artisan migrate command. This will migrate the tables to the database.
* To rollback a migration we can use the php artisan rollback command.

php artisan migrate:rollback

This will delete the last migration that we did.

**Adding Columns to Existing tables using migrations**

* We can do that by php artisan command line
* We use

php artisan make:migration name\_of\_the\_migration - -table=“table\_name”

This will create a new migration which will reference the post table.

Then in the migration file we make the changes in the function up()

we mention the column to be added like :

$table->integer(‘is\_admin’)->unsigned();

We can chain the constraints like above.

* We can also edit the down() function for dropping the table.

like :

$table->dropColumn(‘is\_admin’);

-After these steps use php artisan migrate

**Some other commands for migrations**

* To reset a migration we use php artisan migrate:reset .

this will remove all the created tables from the database.

* To refresh the migration we use

php artisan migrate:refresh.

What this does is it will rollback the migration and migrate it again.

* To check the status of the migration we use

php artisan migrate:status

This will show a table with the list of all migrations and show whether it ran or not.

**Running queries in the database**

**Raw SQL Queries**

* Using facade we can insert data directly into the database.
* Eg:

DB::insert(‘insert into posts (title, content) values(?,?)’,[‘php with Laravel’,’Laravel is the best thing that has happened to php’]);

Here we use PDO for database operations. We add pass values to the query through arrays. This can be directly done from routes.php with the help of a helper function in a route.

* In case of reading data using raw queries.

eg:- $results = DB::select(‘select \* from posts where id = ?’,[2]);

This will return the selected record as an object of stdClass. We can get each value by iterating over the object using foreach loop.

Like

foreach($results as $posts){

return $posts->title;

}

Or you can directly return the object without using foreach loop.

* For updating data we can do similarly.

eg ;- $affected = DB::update(‘update posts set title=?,updated\_at=? Where id = ?’,[‘updated title’,’NOW(),2]);

return $affected;

This will update the title and updated\_at field of posts where id = 2. We can set values for multiple columns in , separated way. The $affected will have the value of number of rows in which updation has taken place.

* Delete operation is similar to the above mentioned other operations. Example:- $deleted = DB::delete(‘delete from posts where id = ?’,[2]);

**Eloquent in Laravel(Object Relational Mapper(ORM))**

* it lets you to create database queries in a much easier manner. It is one of the most important features of Laravel.

**Model**

* To create a model we use php artisan make:model model\_name -m

the -m flag creates a migration for the model at the same time.

if you don’t want a migration we can omit it.

Eg:- php artisan make:model Post

* Models will be in the Models folder inside app(app/Models)
* By default Laravel assumes that the name of the model is the same as the table name. It takes the name of model as table name in all lower case. If it is not same we have to tell explicitly that the name of the table is different. It is done by creating a protected variable $table and assigning it to the table name.

for example:- class PostAdmin extends Model{

protected $table = ‘posts’;

}

* Also by default Laravel assumes that the primary key of the table is id. If not we have to explicitly specify the primary key like we did in the case of table name.

ie protected $primaryKey = ‘primary\_key’

* If both of the above are not in the case we don’t need to write them down in our model.
* To use a model we have to import it. It can be done in 2 ways.

-> by defining “ use App\Models\Model\_name “ in the beginning of the php file

-> by assigning it to a variable ie, “$model = App\Models\Model\_name”

**Note the use App\Models**

* We can pull all the records from the table by using Model\_Name::all()

eg :- $posts = Post::all()

Ensure that we import the Model Post by using any of the above steps

* To find a particular record from the table we can use find() by passing the primary key value.

eg:- $post = Post::find(2);

will return the record and store it in $post as an object.

**Retrieving data with constraints**

* where() function of the model is used for this.
* Eg:- $posts = Post::where(‘id’,2);

here ‘id’ is the primary key of the table post.

* We can chain multiple constraints together using ->
* Eg:- $posts = Post::where(‘id’,2)->orderBy(‘id’,desc)->take(1)->get();
* The findOrfail() is another method for finding a record, but if the record is not found it will raise an exception.

eg:- $posts = Posts::findOrfail(1);

the parameter passed to this method is the primary key.

* We can use where() in multiple ways

eg:- $posts = Posts::where(‘users\_count’,’<‘,50)->firstOrfail();

here we use conditions inside the where separated by ,.

the first() method is used to get the first record. If we use firstOrfail() it will return the first record or it will fail.

**Inserting / Saving Data with Eloquent**

* To insert data to a table /model. First create an instance of the model using new. Then access each column as an object of the instance and set values to each column using ‘=‘.

Eg:-

$post = new Post;

$post->title = ‘New eloquent title insert’;

$post->content = ‘The content’;

* To actually insert the data into the table use $obj\_name.save(). This will insert the data.

For the above example we can use $post->save(); which will insert the content.

* To make use of the save() to update an existing record we only need to make a small change. Ie, instead of creating a new instance of the model use the find() static method to find the value. Then like we inserted data we can set values to the columns and after that we can call the save() method.

Eg:- $post = Post::find(2);

$post->title = ‘Some title’;

$post->content = ‘The content’;

$post->save();

**Creating Data (Concept of mass assignement)**

* This works best when used with forms. When you create data there is something called a mass assignment operation that will go on. Laravel prevents multiple columns from being created at the same time unless you specify explicitly in the model.
* To test this we use the create() method of the model. Inside the argument of the create method we pass the column names and values as associative arrays.

Eg:- Post::create([‘title’=>’the create method’,’content’=>’I\’m learning laravel’]);

Note that we don’t need to assign this to a variable. Also special characters such as single-quotes are escaped using \.

* If we use the above statement we will get a mass assignment exception. To overcome this we have to define the $fillable array and set the name of columns that we can do mass assignment on. Eg:-

class Posts extends Model{

protected $fillable = [‘title’,’content’];

}

This will enable the mass assignment functionality to work.

**Updating Data With eloquent**

* We can update data with eloquent in other ways also.

Eg: Route::get(‘/update’,function(){

Post::where(‘id’,2)->where(‘is\_admin’,0)->update([‘title’=>’New Title’,’content’=>’I like Laravel’]);

});

The update() here is used to update the title and content.

**Deleting Data with eloquent**

* We can delete a record by finding it using find method with the primary key passed as an argument to the find() method. The found record if assigned to a variable(object). On this object delete() method is called.

Eg:- $post = Post::find(3);

$post.delete();

* Another way is by calling destory(‘primary\_key’) directly on the model.
  + - Example: - Post::destroy(1);
* If you want to delete multiple records, we can pass an array of primary keys as arguments for the destroy() method.

Eg: - Post::destroy([4,5]);

**Soft Deleting or Trashing**

* We can also delete items in such a way that it is not permanently deleted.
* In-order to implement this we need to import the SoftDeletes in the Model

for that use Illuminate\Database\Eloquent\SoftDeletes;

Then inside the model we mention

use SoftDeletes;

And use protected $dates = [‘deleted\_at’];

This is used to identify when the item is soft deleted. This defines a column ’deleted\_at’ in the table. This column is set by Laravel.

* Then create a new migration using
* php artisan make:migration adding\_softdelete —table=posts

where adding\_softdelete is the name of the migration and posts is the name of the table.

* The next step is to create a table for deletes inside the migrations with

$table->softDelete();

in the up() function

Incase you want to drop the column you can add

$table->dropColumn(‘deleted\_at’);

In the down() function.

* If you want to delete a migration manually deleting the file will cause problems so for this we have to first

php artisan migrate:reset which will rollback the migration then we can delete the migration manually.

* then we can delete the record using the delete() method.
* The deleted record cannot be fetched like a normal record.
* To find a trashed record we can use

$post = Posts::withTrashed()->where(‘id’,4)->get();

Which will return all the values(including trashed and not trashed)

another way is to use

onlyTranshed() method

$post = Posts::onlyTrashed()->where(‘id’,4)->get();

Will give only trashed values.

**Retrieving soft deleted items**

* You just need to chain the restore() method to the the reading query used before

Ie, Posts::withTrashed()->where(‘id’,4)->restore();

**Permanently Delete item**

* To permanently delete an item use the forceDelete() chained to the reading query.

Ie, Posts::withTrashed()->where(‘id’,4)->forceDelete();

This will delete the record permanently from the database.